Simulation répartie de systèmes à évènements discrets. Partie 1 : modélisation et schémas d'exécution

Philippe Ingels, Michel Raynal

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1057

Programme 3

SIMULATION REPARTIE DE SYSTEMES A EVENEMENTS DISCRETS : PARTIE 1 : MODELISATION ET SCHEMAS D'EXECUTION

Philippe INGELS
Michel RAYNAL

Juillet 1989

RR_1057

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Simulation répartie de systèmes à événements discrets : Partie 1 : Modélisation et schémas d'exécution *

Philippe Ingels Michel Raynal

Publication Interne n° 476 - 26 Pages
Juin 1989

I.R.I.S.A.
Campus de Beaulieu
35042 RENNES CEDEX
E-mail: raynal@irisa.fr

Simulation répartie de systèmes à événements discrets : Partie 1 : Modélisation et schémas d'exécution

Résumé

La simulation répartie à événements discrets est un sujet intéressant à la fois du point de vue de l'algorithmique répartie, car on y retrouve un certain nombre de problèmes-type, et du point de vue de l'expérimentation en vraie grandeur de programmes parallèles, car tout gain de temps peut être déterminant pour la viabilité de certaines simulations. On examine dans ce rapport d'une part la modélisation des systèmes à événements discrets dans un contexte de répartition (modèle à base de processus et de messages), et d'autre part les problèmes posés par la mise en œuvre de simulateurs répartis.

Un simulateur est un interpréteur de temps virtuel (le temps de la simulation). Réaliser un simulateur réparti consiste donc à définir un schéma d'exécution distribué qui assure la progression du temps virtuel (vivacité) en respectant les relations de causalité du système simulé (sureté). Une présentation des principaux types de simulateurs est effectuée dans le cadre synchrone (dirigé par le temps) et asynchrone (dirigé par les événements).

Distributed simulation of discrete event systems : part 1 : Models and algorithms

Abstract

Distributed simulation of discrete event systems is an interesting topic from the point of view of both distributed computing (it presents some paradigms of distributed computing) and experimentation of parallel programs. This report provides an overview first of the modelling of discrete event systems in a distributed context, and second of problems encountered to implement distributed simulations.

A simulator is an interpreter of virtual time (the one of the simulated system). Implementation of distributed simulator lies in defining a distributed run-time which ensures the progress of this virtual time (liveness) without violating causal dependencies of the simulated system (safety). A presentation of the major types of these interpreters is given in synchronous (time-driven) and asynchronous (event-driven) contexts.

*Ce travail a été en partie supporté par le Greco C3

Table des matières

1 Introduction

La synthèse présentée est relative à la mise en œuvre distribuée de simulation à événements discrets. On n'examine pas ici la simulation continue, c'est à dire le cas des systèmes modélisés par des équations aux dérivées partielles (problèmes d'avionique, de météorologie, etc...); dans ce cas la solution repose sur un schéma d'exécution itératif (dont la mise en œuvre sur des machines parallèles peut toutefois s'avérer complexe).

Ce travail répond à une double motivation: d'une part étudier l'adéquation d'une classe de machines (les multiprocesseurs à communication par messages) à une classe d'application (la simulation distribuée) et d'autre part analyser une application où on retrouve les principaux problèmes posés par l'algorithmique distribuée.

Si les "types extrêmes" de calculateurs parallèles ont des domaines d'application relativement bien définis - d'une part les calculateurs vectoriels à mémoire commune qui paraissent bien adaptés aux traitements matriciels, et d'autre part les réseaux généraux ou locaux qui semblent répondre aux besoins des systèmes répartis (base de données, bureautique,...), - il reste à dégager les classes d'applications qui conviennent aux multiprocesseurs à communication par messages (dont l'*iPSC* est un exemple). La simulation distribuée est un domaine d'application qu'il peut être intéressant de mettre en œuvre sur ce type de machine; certaines simulations à événements discrets sont en effet par nature non vectorielles; et même si la parallélisation d'une simulation peut n'offrir qu'un gain réduit [27], la complexité des systèmes simulés est actuellement telle (certaines simulations peuvent demander des milliers d'heures de calcul) que le moindre gain de temps de simulation est appréciable [31].

Dans la section 2 on étudie la problématique de la simulation à événements discrets : le choix d'un modèle pour représenter le système réel, son expression sous forme d'un programme, et enfin les problèmes posés par la réalisation d'un interpréteur pour la simulation à événements discrets (le simulateur); il s'agit essentiellement de la représentation et de la gestion du temps simulé.

La section 3 aborde les problèmes propres à la mise en œuvre distribuée d'une simulation. On présente le problème fondamental: le respect dans l'interprétation du modèle des relations de causalité existant dans le système réel, et les deux grandes approches possibles : la simulation dirigée par le temps ou la simulation dirigée par les événements. Dans le premier cas concevoir un simulateur revient à définir un synchroniseur [3], [2].Dans le second cas on étudie les deux grandes classes de solution : la solution optimiste dans laquelle on laisse évoluer le système sans contrôle jusqu'à ce qu'il y ait une violation du principe de causalité, ce qui nécessitte un retour en arrière [21]; et la solution pessimiste qui évite les violations du principe de causalité, mais peut conduire à un interblocage qu'il faut prévenir [29], [14] ou détecter et guérir [12].

Un second rapport viendra compléter celui-ci en donnant des résultats d'expérimentations de schémas d'exécution relatifs à la simulation distribuée sur l'hypercube d'Intel.

2 La simulation à événements discrets

La simulation consiste à effectuer des expériences sur un modèle d'un système réel, cette activité peut se décomposer en quatre étapes :

1. Il faut tout d'abord analyser le système réel et les informations que l'on désire obtenir sur son comportement, pour élaborer ensuite un *modèle* de ce système réel dans lequel on ne retient que les caractéristiques qui semblent pertinentes par rapport aux résultats cherchés.

2. L'étape suivante, pour une simulation informatique, consiste à représenter ce modèle sous forme d'un *programme de simulation.*

3. On aborde ensuite la phase d'expérimentation dans laquelle on fait évoluer le modèle, c'est à dire que l'on interprète le programme de simulation à l'aide d'un schéma d'exécution adapté (le *simulateur*).

4. la dernière étape consiste à analyser les résulats produits lors de la phase précédente, pour en déduire des informations sur le comportement du système réel.

L'activité de simulation consiste en fait en un va et vient entre ces quatre phases : l'analyse des résultats peut conduire à effectuer de nouvelles expériences, ou à remettre en cause le modèle parce qu'il n'est pas assez précis, ou qu'il ne capte pas bien les caractéristiques intéressantes du système réel.

Dans le paragraphe §2.1 on précise les modèles auxquels on s'intéresse et leurs propriétés, et en §2.2 on aborde la description des programmes de simulation. Le paragraphe §2.3 décrit les éléments essentiels de la réalisation d'un simulateur séquentiel.

2.1 Le système réel et son modèle

Les systèmes réels auxquels on s'intéresse obéissent à deux principes :

1. *principe de causalité* : le futur ne peut influencer le passé. Plus précisement, l'état du système à l'instant t est indépendant de tout ce qui peut se produire à une date t' supérieure à t.

2. *principe de déterminisme* : le futur du système peut être déterminé à partir de son état présent et de son passé. En d'autres termes, à tout instant t il existe une valeur ϵ telle que le comportement futur du système est prévisible jusqu'a $t + \epsilon$.

Ces deux principes qui gouvernent l'évolution des systèmes réels étudiés portent sur le temps (causalité) et la loi d'évolution du système (déterminisme). Toute modélisation devra respecter le principe de déterminisme et tout simulateur devra garantir le principe de causalité.

On considère les systèmes réels qui peuvent avoir une modélisation discrète. Un tel système est caractérisé à un instant donné par un état, et son comportement au cours du temps peut être décrit par une séquence de transitions d'états (correspondant à des *événements*). Il peut être décomposé en entités actives dotées chacune d'un état et d'un comportement propres, ces entités interagissant pour produire le comportement global du système.

Entités actives, temps, état et transitions d'état sont donc les éléments clés pour modéliser un tel système. Le concept de processus est un outils efficace pour représenter le comportement d'entités actives, c'est pourquoi nous parlerons de *processus du modèle* (ou Pm). Les interactions

entre processus peuvent être représentés de diverses manières, elles impliquent toujours un échange d'information (celle-ci pouvant n'être qu'un simple signal). A la suite de Peacock [29], Chandy et Misra [14] proposent de modéliser ces interactions par des échanges de messages, ce qui assure d'une part une bonne capacité d'expression pour le modèle, et d'autre part une transposition aisée dans le contexte distribué.

Un modèle sera constitué d'un graphe de processus, deux processus du modèle étant reliés si ils décrivent des entités interagissant dans la réalité. Les processus du modèle interagissent en échangeant des messages, et la communication d'un message y prend un temps nul (en effet l'interaction met en jeux les deux entités du système réel simultanément).

Si par exemple on veut modéliser une station de lavage de voiture, où les voitures arrivent, attendent leur tour (dans l'ordre d'arrivée), se font laver (ce qui prend TL unités de temps) puis quittent le système, on peut adopter le modèle suivant, comportant 3 processus :

- *source* : représente l'arrivée des voitures à des intervalles de temps suivant une certaine loi.
- *laveur* : représente le laveur lui-même.
- *sortie* : matérialise la sortie des voitures du système.

La figure 1 représente les communications existant entre les processus du modèle. L'émission d'un message de la *source* vers le *laveur* correspond à l'arrivée d'une voiture dans le système. L'émission d'un message du *laveur* vers la *sortie* correspond à la fin du lavage d'une voiture. Le processus *laveur*, d'une part réceptionne les voitures (ie. messages venant de la *source*) et les range dans une file, et d'autre part prend les voitures dans l'ordre où elles se présentent dans la file et les lave.

Figure 1 : modèle de la station de lavage de voitures

On ne considère dans le modèle que les interactions directes. Si le système réel contient deux entités, interagissant via un medium physique sur lequel les temps de transit sont non nuls, on doit se ramener à des interactions directes entre *Pm*. Si on reprend l'exemple du lavage de voiture, dans le cas où avant d'aller vers le laveur il faut prendre un jeton à la caisse, le trajet caisse/laveur prenant un temps δ, on a trois solutions pour modéliser ce système:

- Avoir deux processus (*caisse* et *laveur*), et incorporer le temps de transit dans l'activité du processus *caisse*. C'est à dire que que le temps de service à la caisse est augmenté de δ.

- Avoir toujours deux processus (*caisse* et *laveur*), mais incorporer le temps de transit dans l'activité du processus *laveur*.

- Avoir trois processus (*caisse*, *chemin* et *laveur*), le processus *chemin* recevant une voiture (message) venant de la caisse à t, envoie cette voiture vers le *laveur* à $t + \delta$.

Il faut noter que les principes auxquels obéissent les systèmes réels étudiés (causalité et déterminisme) ont des conséquences sur leur modélisation, c'est à dire sur la forme du réseau de processus constituant le modèle et sur ses règles d'évolution. Celui-ci doit satisfaire les deux conditions suivantes [28] :

1. *condition de réalisabilité* : un message envoyé par un Pm à l'instant (simulé) t est une fonction de son état initial et des messages qu'il a reçu jusqu'à t (inclus). En d'autres termes un message reçu après t ne peut avoir d'effet sur le comportement du Pm jusqu'à t (cette condition découle du principe de causalité).

2. *condition de prédictabilité* : si il existe un circuit sur le réseau des Pm, alors à tout instant t il existe un Pm de ce circuit tel que les messages envoyés par ce processus peuvent être calculés jusqu'à $t + \epsilon$ (avec $\epsilon > 0$) en fonction de son état initial et des messages qu'il a reçu jusqu'à t inclus. En d'autres termes, sur un circuit il existe à tout instant au moins un processus ayant un temps de service non nul, c'est à dire tel que la réception d'un message à t provoquera l'émission d'un message à t' avec $t' > t$. Cette condition sur le modèle découle du principe de déterminisme imposé au système réel.

2.2 Le programme de simulation

Une fois le modèle du système réel conçu il s'agit de le décrire par un programme. Le problème principal posé par cette description est celui de la manipulation du temps simulé. Deux types d'expression sont possibles :

- par *événements*, chaque événement est décrit d'une part par une condition (la condition d'occurence de l'événement), et d'autre part par les instructions qui représentent le changement d'état correspondant. Le programmme est alors composé de la description des structures de donnée représentant l'état de chaque entité du système, et de la description de tous les événements possibles. Les actions associées à l'occurence d'un événement sont toujours de durée nulle dans le temps simulé.

- par *processus*, on représente alors dans le programme de simulation les processus du modèle par des processus qui exécutent des actions matérialisant les changements d'état, l'évolution du temps et les interactions entre processus; dans ce cas certaines de ces actions ont une durée non nulle dans le temps simulé.

Dans le cas d'une description par événement, le modèle du processus *laveur* de le figure 1 (la description des processus *source* et *sortie* n'est pas donnée ici) peut être décrit comme suit (*Tsim* représente la valeur courante du temps simulé) :

tfin : temps
fv : file de voitures

{événement "arrivée de voiture"}
QUAND recoit *m* de *source*
  FAIRE
        SI *fv* vide
        ALORS *tfin*:=*Tsim*+*durée-lavage* FSI
        mettre *m* dans *fv*
  FAIT

{événement "fin de lavage"}
QUAND *Tsim*=*tfin*
  FAIRE
        m:=tête de *fv*
        envoyer *m* à *sortie*
        SI *fv* non vide
        ALORS *tfin*:=*Tsim*+*durée-lavage* FSI
  FAIT

Dans le cas d'une description sous forme de processus, les interactions entre processus du programme de simulation peuvent être de différentes natures :

- *interaction par ressource*, une ressource est un objet passif utilisé par les processus du programme de simulation. Le langage Simone [23] est un exemple de langage permettant ce type de description, il s'agit de la modélisation connue sous le nom de "client-serveur" [8].

- *interaction directe*, les processus du programme de simulation peuvent s'activer ou se désactiver directement. Simula [16] est l'illustre représentant de cette classe.

- *interaction par message*, les processus du programme de simulation émettent et recoivent des messages. Le langage May [4] constitue un bon représentant de cette classe de langages.

Ce dernier type de langage est particulièrement intéressant dans notre cas; le langage correspond bien à la fois aux modèles que nous avons définis, et au support d'exécution que nous visons. Il faut préciser la sémantique des émissions et réceptions de messages : les messages reçus par un processus sont stockés dans une file, et délivrés au fur et à mesure des demandes. Ainsi un message émis par p_i vers p_j a une date t (dans le temps simulé), est disponible à t dans la file de réception de p_j. Sa consommation effective par p_j, à sa demande, sera effectuée à une date t' supérieure ou égale à t.

Un processus du programme de simulation peut donc exécuter quatre types d'actions :

- La mise à jour des variables décrivant son état. Ces actions ont une durée nulle dans le temps simulé.

- La matérialisation de l'écoulement du temps simulé (correspondant à la durée des actions dans la réalité). Cette synchronisation par rapport au temps simulé peut se traduire par la construction *attendre*($Tsim = t$), ou $Tsim$ représente la valeur du temps simulé.
- L'émission d'un message, cette action est de durée nulle dans le temps simulé.
- La synchronisation avec d'autres processus, représentée par l'attente d'un message. Cette action à une durée éventuellement non nulle, et non connue a priori, dans le temps simulé. Elle peut se traduire par une construction de la forme *attendre*(*message m*).

Dans ce cadre le processus *laveur* du modèle de la station de lavage de voitures, peut être décrit par un processus simulé ayant la forme suivante:

PROCESSUS laveur;
  REPETER
    attendre(message *voiture*);
    attendre($Tsim = Tsim +$ *durée_lavage*);
    émettre *voiture* vers *sortie*;
  FIN REPETER

Il faut noter que le schéma de communication avec tamponnage des messages permet de décrire facilement le comportement du *laveur*, en particulier pour représenter le fait qu'une voiture arrivant alors que le *laveur* est occupé est mise en attente par celui-ci. Cela est du au fait que le *laveur* (processus du modèle) n'est pas un processus au sens informatique du terme : il est capable de faire plusieurs choses en même temps. Une communication par rendez-vous nécessiterait d'introduire dans le modèle un processus supplémentaire représentant la file d'attente.

2.3 La simulation d'un modèle à interaction par messages

2.3.1 Interprétation d'un programme de simulation

On a vu précédement (paragraphe 2.1) que le résultat de la modélisation est un réseau de processus communiquant par messages (nous désignons par Pm ces processus du modèle). Nous avons également vu que ces processus évoluent dans le temps et que les communications de messages (interactions) sont de durée nulle. Ceci est vrai quel que soit les outils utilisés pour décrire ces Pm sous forme de programme informatique.

Interpréter le programme de simulation consiste à l'exécuter de façon à ce que son temps de référence soit le temps simulé (et non le temps de la machine supportant l'exécution), ceci inclut d'une part l'avancement du temps et d'autre part le respect des relations de causalité. Nous appelerons *processus simulé* (Ps) la suite d'actions exécutée sur le calculateur pour représenter l'évolution d'un processus du modèle. La simulation est dite correcte si il lui est possible de prédire grâce aux Ps les séquences exactes de messages communiqués entre Pm, et les dates associées [28].

Le problème fondamental dans la conception d'un simulateur est donc la gestion du temps simulé. Il s'agit d'en trouver une représentation et de construire le schéma d'exécution qui en assure une gestion correcte. Deux grandes classes de solutions ont été proposées pour cela. Dans les deux cas il existe une variable accessible en lecture à tous les Ps qui fournit la date courante ($Tsim$):

1. *Simulation conduite par le temps* : le simulateur avance la variable $Tsim$ par pas de dt. A chaque incrémentation il examine les Ps et exécute les actions impliquées par leur état et la valeur courante de l'horloge.

2. *Simulation conduite par les événements* : le simulateur évalue le premier instant $tsuiv$ dans le futur où se produira une modification de l'état du système(événement), affecte à $Tsim$ la valeur $tsuiv$ et effectue les actions correspondant à l'événement sélectionné. Ces actions peuvent produire de nouveaux événements pour le futur, où en supprimer.

La simulation conduite par le temps est intéressante lorsqu'il y a beaucoup d'événements à simuler par incrément de temps, de plus cet incrément de temps peut être rendu variable au cours de la simulation. La simulation par événements est par contre plus souple, toute action du simulateur fait en effet progresser la simulation.

2.3.2 La simulation séquentielle

Réaliser une simulation séquentielle des systèmes à évolution discrète est simple au niveau des principes. Dans le cas d'une simulation dirigée par le temps, le schéma d'exécution assure que tout événement survenant à t dans le temps simulé sera exécuté avant (dans le temps réel) un événement survenant à $t + k * dt$ pour tout $k > 0$.

Dans une simulation dirigée par les événements un message m émis par un Pm à l'instant simulé t est représenté par un couple (t, m). On gère une liste de tels couples, correspondant à toutes les interactions futures prévues à partir de la simulation du système jusqu'à la date courante. Il s'agit des interactions qui seront produites par chaque Pm si d'ici leurs dates d'occurence il ne se passe rien de nouveau dans le Pm émetteur. Cette liste est triée par ordre de dates d'occurence croissantes. Le simulateur considère alors le premier couple (t, m) et effectue les actions suivantes :

- met à jour l'horloge ($Tsim$) à la valeur t.

- simule l'effet de la réception de m à $Tsim$; cela peut ajouter ou supprimer des éléments de la liste, ces éléments sont tels que leurs dates d'occurence sont supérieures où égales à $Tsim$.

Nous renvoyons le lecteur à [28] qui donne une preuve de correction de cet algorithme de simulation séquentielle, dans le cas d'un modèle à interactions par messages.

3 La simulation distribuée

3.1 Le contexte distribué

Le support d'exécution de la simulation est constitué par un ensemble de processeurs asynchrones qui peuvent s'échanger de l'information par l'intermédiaire de messages, le transfert d'un message d'un processeur à un autre prenant un temps fini mais arbitraire. L'hypercube *iPSC* d'INTEL est un exemple d'une telle machine multiprocesseurs, à communication par messages. Il faut alors résoudre deux problèmes pour réaliser une simulation distribuée sur une telle machine:

1. Le placement des processus du programme de simulation sur les processeurs de la machine.
2. La construction d'un schéma d'exécution qui d'une part rende compte du temps simulé (ie. garantisse sa progression), et d'autre part assure que les processus du modèle sont exécutés correctement par rapport au temps simulé (ie. les relations de causalité entre les événements sont bien respectés).

Le problème du placement n'est pas abordé ici. Le lecteur peut se reporter à [18] (chapitre 5). Notons toutefois que la création dynamique de processus peut être nécessaire dans un programme de simulation et qu'en conséquence l'algorithme de placement et d'ordonnancement des tâches doit en tenir compte.

Le second problème concerne la distribution d'un temps simulé global. On affecte pour cela une horloge logique locale à chaque processus simulé; le rôle de celle-ci est de représenter l'écoulement du temps simulé global. Afin que l'ensemble de ces horloges locales représentent correctement le temps simulé il est nécessaire de synchroniser leurs évolutions. Deux classes de solutions sont possibles, elles se distinguent par la synchronisation plus ou moins forte qu'elles imposent sur l'évolution des horloges. Il s'agit des versions distribuées des simulations dirigées par le temps et par les événements.

Il faut noter qu'il y a une différence entre l'ordre chronologique des événements et les relations de causalité entre événements (au moins pour des événemnts apparaissant dans des processus différents). La relation de causalité impose un ordre partiel sur les événements : si e_1 est directement ou indirectement la cause de e_2 alors la simulation de e_1 doit précéder la simulation de e_2. Il n'est pas exclu que deux événements survenant au même temps simulé soient liées par un relation de causalité : c'est le cas par exemple d'un processus qui recevant un message à t, réagit en émettant instantanément (dans le temps simulé) un autre message. Par contre si deux événements ne sont pas liés par une relation de causalité, leur simulation peut se faire dans un ordre quelconque. Ainsi la condition pour que l'on puisse simuler de façon indépendante (par exemple en "même temps") deux événements e_1 et e_2, est que

$$e_1 \textit{ et } e_2 \textit{ ne soient pas liés par une relation de causalité,}$$

et non pas que e_1 et e_2 se produisent au même temps simulé.

3.2 La simulation distribuée dirigée par le temps

Dans ce cas les horloges locales progressent de concert. Lorsqu'elles ont la valeur t les processus exécutent les événements correspondants, après quoi les horloges prennent la valeur $t + \delta$. En d'autres termes l'horloge globale du temps simulé est distribuée par duplication dans chacun des processus de la simulation. Plusieurs mises en œuvre sont possibles, nous en citons deux.

Dans une première approche chaque processus, en fonction de son comportement, cherche à faire progresser son horloge locale. Un algorithme de contrôle est alors placé sur l'ensemble de ces horloges locales pour que leurs dérives mutuelles n'excèdent pas δ [30] (chapitre 2).

Une seconde mise en œuvre possible repose sur le concept de synchroniseur introduit par Awerbuch [3]. Un synchroniseur est un interpréteur d'algorithme distribué synchrone sur un réseau asynchrone. Un algorithme distribué synchrone est conçu pour s'exécuter sur un réseau de processus tel que :

- L'évolution des processus est cadencée par une horloge globale, un cycle de cette horloge est nommée *pulsation*.
- Le délai de transfert d'un message est inférieur à une pulsation (un message émis au début d'une pulsation est reçu et traité par son destinataire avant le début de la pulsation suivante).

Plusieurs synchroniseurs sont envisageables, ils se distinguent par les mécanismes de synchronisation utilisés (qui réalisent un contrôle plus ou moins centralisé) et par leurs complexités en nombre de messages et en temps. Ces algorithmes maintiennent sur chaque processus du réseau asynchrone une représentation locale de l'horloge globale du réseau synchrone; suivant le synchroniseur choisi ces valeurs possèdent l'une ou l'autre des proprietés suivantes :

- *Synchronisme fort* : la différence entre le numéro de pulsation de deux processus quelconques du réseau reste toujours inférieure ou égale à 1.
- *Synchronisme faible* : la différence entre le numéro de pulsation de deux processus du réseau séparés par p liens reste toujours inférieure ou égale à p.

On trouvera dans [2] une caractérisation précise des propriétés de ces synchroniseurs et dans [1] une étude expérimentale de leurs efficacités respectives. L'algorithmique distribuée sous-jacente à ces interpréteurs est l'algorithmique des phases [19], chapitre 4.

Le simulateur peut être décrit comme un algorithme synchrone dans lequel à chaque pulsation on effectue la simulation corespondant à un nouvel incrément de temps δ ($Tsim_i$ représente la valeur courante du temps simulé pour le processus i) :

{Pour le processus P_i}
QUAND signal de pulsation
FAIRE
  $Tsim_i$:=$Tsim_i + \delta$
  simuler le processus p_i en fonction de son état courant et de $Tsim_i$

(ceci inclut l'émission de messages)
FAIT

QUAND reçoit message m
FAIRE
mettre à jour l'état courant de p_i en fonction de m
FAIT

Pour interpréter un tel simulateur il suffit de disposer d'un synchroniseur assurant un synchronisme faible. En effet, prenons les trois processus p_i, p_j et p_k de la figure 2. Si p_i commence la pulsation p ($Tsim_i = p * \delta$) c'est qu'il est sûr d'avoir reçu tous les messages émis à la pulsation $p-1$, et donc que p_j à émis tous les messages relatifs à la pulsation $p-1$. De même puisque p_j a démarré sa pulsation $p-1$, p_k à émis tous les messages concernant la pulsation $p-2$. Donc les messages suivants émis par p_k vers p_j le seront à la pulsation $p-1$ ($Tsim_k = (p-1) * \delta$,) et en réaction p_j peut émettre un message vers p_i à $Tsim_j = p * \delta$. La réception de ce message par p_i à la pulsation p ($Tsim_i = p * \delta$) est correcte et ne viole pas le principe de causalité.

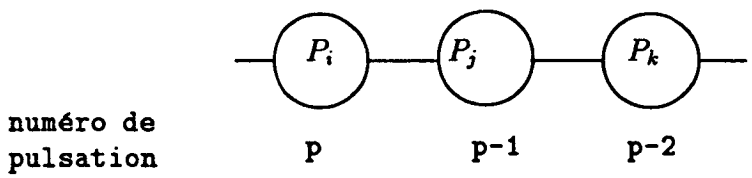

Figure 2 : Exemple de processus synchrones

Il faut noter que, dans un algorithme synchrone, la réception d'un message à la pulsation p ne peut pas provoquer l'émission d'un message à cette même pulsation. Il n'est donc pas possible de simuler des processus ayant des réactions instantannées, puisque le transfert d'un message du programme de simulation consomme par définition δ unités de temps simulé (une pulsation).

3.3 La simulation distribuée dirigée par les événements

3.3.1 Le problème posé et les classes de solutions

Dans la simulation dirigée par les événements le simulateur doit déterminer quel est le prochain instant, dans le temps simulé, où va se produire un changement d'état dans le système, puis modifier en conséquence le temps simulé et effectuer le traitement décrivant le changement d'état. Dans un contexte distribué il y a un simulateur par processeur, et il ne connait pas l'état global du système. Il doit prendre sa décision sur l'avancement du temps simulé à partir

d'informations locales, éventuellement calculées grâce au contenu des messages qu'il a reçu des autres processus.

Chaque processus simulé Ps_i va être doté d'une variable locale $Tsim_i$, mise à jour par le simulateur, et représentant la date courante dans le temps simulé. L'émission d'un message m par un processus du modèle Pm_i, vers Pm_j, à l'instant simulé t, va être traduite par l'émission d'un message estampillé (t, m) de Ps_i vers Ps_j. La prise en compte par Ps_j du message (t, m) provoque l'affectation à $Tsim_j$ de la valeur t.

Le problème central est celui du respect du principe de causalité. Si un processus Ps_i prend en compte un message estampillé t' après un message estampillé t, avec $t' < t$, ce principe est violé; le futur (message estampillé t) a pu influencer le passé (traitement du message estampillé t').

Définition 3.1 *La contrainte de causalité locale est dite satisfaite par un processus simulé si et seulement si les messages reçus par ce processus sont traités dans l'ordre non décroissant des estampilles.*

Dans [18] il est montré de plus que :

Propriété 3.1 *Si tous les processus simulés respectent la contrainte de causalité locale, et s'ils n'interagissent que par échanges de messages estampillés, alors la simulation dans son ensemble respecte la condition de réalisabilité (cf. 2.1).*

Afin de garantir le respect de ces contraintes par le simulateur distribué deux classes de solutions ont été proposées :

1. La *stratégie optimiste (ou cohérente a posteriori)* dans laquelle les messages sont traités dans l'ordre d'arrivée jusqu'à ce que l'on détecte une violation de la contrainte de causalité locale. Dans ce cas il s'agit d'une erreur dans l'interprétation de la simulation, qui nécessite un retour en arrière. Il s'agit d'une stratégie basée sur la notion de *temps virtuel* décrite par Jefferson [21].

2. Les *stratégies pessimistes (ou cohérente a priori)* dans lesquelles on met en œuvre un algorithme d'interprétation de la simulation assurant que la contrainte de causalité locale sera toujours respectée.

3.3.2 Les stratégies cohérentes a priori : risque d'interblocage

Nous supposons que les canaux reliant les processus simulés sont FIFO et sans perte de message.

Définition 3.2 *Chaque processus simulé détermine sur chacun de ses canaux (entrant et sortant) un temps-canal calculé comme suit :*

- *Le temps-canal d'un canal sortant est l'estampille du dernier message émis sur ce canal.*
- *Le temps-canal d'un canal entrant est soit l'estampille du premier message à consommer sur ce canal s'il n'est pas vide, soit l'estampille du dernier message recu et consommé sinon.*

- *Initialement le temps-canal de tous les canaux vaut 0.*

Soit *Tmin* le minimum des temps-canal de tous les canaux entrants d'un processus. S'il existe un canal entrant *non vide* dont le temps-canal est *Tmin*, le processus peut traiter le message présent sur ce canal car il est sûr de ne jamais recevoir un message ayant une estampille inférieure (car les canaux sont FIFO). Mais si tous les canaux de temps-canal *Tmin* sont vides le processus doit attendre, car il est possible qu' il recoive sur ces canaux des messages ayant une estampille inférieure à celle des messages actuellement présents sur les autres canaux.

De plus il n'est pas certain que l'évolution ultérieure de la simulation le débloque. Prenons comme exemple le modèle de la figure 3, dans lequel p_1 après réception d'un message de p_0, émet un message soit vers p_2 soit vers p_3 en fonction d'une certaine règle. Si au cours d'une simulation aucun message n'est envoyé vers p_2, le temps-canal du canal (p_2, p_4) restera égal à 0 et le processus p_4 ne pourra pas faire avancer son temps simulé. Il restera donc bloqué indéfiniment en attendant l'arrivée d'un message sur le canal venant de p_2, bien qu'il ait des messages en attente en provenance de p_3.

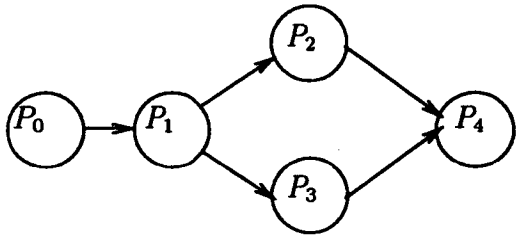

Figure 3 : Réseau pouvant conduire à un interblocage

Il faut noter que cette situation d'interblocage ne correspond pas à un interblocage du modèle ou du programme de simulation, mais résulte de la mise en œuvre du schéma d'exécution distribué destiné à garantir le principe de causalité (rappelons que grâce au principe de prédictabilité le modèle peut toujours évoluer). Pour faire face à une situation d'interblocage il y a deux solutions classiques :

- *La prévention* : dans ce cas un algorithme de contrôle vient s'adjoindre au simulateur pour éviter l'apparition de la situation décrite précédement. Basée sur l'émission de messages supplémentaires (messages NULL), cette solution a été décrite en particulier par Chandy et Misra [14].

- *La détection et la guérison* : ici le simulateur interprète le programme de simulation pour faire progresser tant que cela est possible, (en respectant les contraintes de causalité), les représentations locales du temps simulé (ie. l'algorithme ci-dessus sans les messages NULL). Un algorithme distribué détectant l'interblocage et le traitant en choisissant un processus à relancer vient s'exécuter lorsque les processus de la simulation sont bloqués. Cette solution est présentée dans [12].

3.3.3 Une stratégie cohérente a priori avec prévention de l'interblocage

L'idée de base de cette solution est de faire émettre par les processus simulés, en plus des messages de la simulation, des messages de contrôle (messages NULL) également estampillés par la date (dans le temps simulé) de leur émission. Ces messages NULL ne correspondent à aucun message des processus du modèle. Leur traitement par le processus récepteur n'implique aucun changement d'état de la simulation, mais permet de faire évoluer le temps-canal et donc éventuellement le temps simulé du récepteur.

Une solution envisageable pour remédier au blocage de l'exemple précedant consiste à obliger un processus à émettre un message sur tout ses canaux de sortie à chaque étape de la simulation, soit un message de la simulation s'il y en a un, soit un message NULL sinon. Sur l'exemple de la figure 3, p_1 émet un message NULL vers p_2 (respectivement p_3) à chaque fois qu'il émet un message de la simulation vers p_3 (respectivement p_2). Ainsi p_4 recevra régulièrement des messages sur chacun de ses canaux entrants et pourra donc faire progresser son temps simulé.

Malheureusement cette solution simple ne suffit pas pour prévenir l'interblocage dans le cas où il y a un circuit dans le réseau. Prenons l'exemple du réseau de la figure 4. Au temps 5, p_3 émet le message m_1 vers p_4, il émet donc également un message NULL vers p_2, avec la même estampille. Au temps 10, p_1 émet le message m_2 vers p_2. Les deux canaux entrants de p_2 contiennent un message non consommé, p_2 peut donc avancer son temps simulé au minimum des temps-canal des canaux entrants, c'est à dire au temps 5. Mais la consommation du message NULL ne provoque l'émission d'aucun message vers p_3, et le système se bloque après que p_4 ait consommé le message m_1.

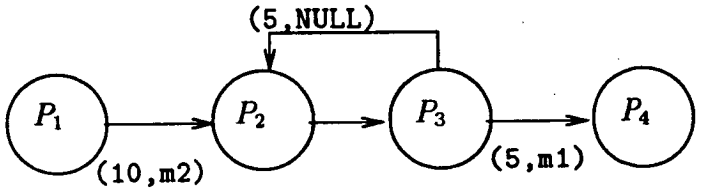

Figure 4 : Réseau contenant un circuit, avec interblocage

Pour résoudre ce problème il faut connaitre une information supplémentaire. Dans la situation de la figure 4 p_2 sait qu'il ne recevra aucun message ayant une estampille strictement inférieure à 5, pour faire avancer la simulation il faut que p_2 puisse déterminer que *dans ces conditions il n'émettra aucun message avant un certain instant* $5 + \delta$ ($\delta > 0$)[1]. Il peut alors communiquer cette information à p_3 grâce à un message $(5 + \delta, NULL)$. Pour un processus cette valeur δ matérialise une *prévision* sur son comportement futur ("look ahead"). Dans notre exemple, si tous les processus p_i sont tels que la prise en compte d'un message $(t, NULL)$ provoque l'émission d'un message $(t + \delta_i, NULL)$ (avec $\delta_i > 0$) sur tous les canaux sortants,

[1] Dans le cas du modèle classique de réseau de files d'attente avec serveur, cette valeur δ est égale au temps de service

alors p_2 finira par recevoir un message $(t', NULL)$ avec $t' > 10$ sur le canal (p_3, p_2), et il pourra alors prendre en compte le message $(10, m_2)$ venant de p_1.

Nous allons décrire plus précisément cet algorithme dans le cadre suivant (résumé dans la figure 5) :

1. On dispose d'un réseau de processeurs ayant la même topologie que le réseau des processus du modèle. Les communications sont asynchrones et les canaux sont FIFO, sans perte de message.

   - *ce* désigne l'ensemble des canaux entrants d'un processeur.
   - *cs* désigne l'ensemble des canaux sortants d'un processeur.
   - A tout canal entrant c est associée une file de messages : $farrivés_c$ où sont stockés les messages au fur et à mesure de leur arrivée. Cette file est illimitée.

2. Sur ce processeur s'exécute un processus simulé représentant le comportement d'un processus du modèle. On peut décomposer ce processus simulé en deux parties logiquement distinctes :

   - La *partie simulation*, qui réalise le travail de simulation proprement-dit (faire évoluer l'état du processus en fonction des messages consommés, envoyer de nouveaux messages). Cette partie est décrite par une procédure, appelée *simuler* :

     PROCEDURE *simuler* = ENTREE(*estampmin*:temps)
     RESULTAT(*datemin*:[*cs*]temps)

     Elle fait évoluer l'état du processus (y compris la représentation locale du temps simulé : $Tsim$) en fonction de l'état courant du processus et des messages figurant dans les files $fdélivrés_c$; *estampmin* est telle qu'il n'y aura pas de messages délivrés avec une estampille inférieure à *estampmin*; cette simulation cesse quand elle ne peut plus faire progresser cet état parce que l'algorithme du processus du modèle nécessite de recevoir un message qui n'est pas encore disponible. Elle rend en résultat un tableau *datemin*, tel que pour chaque canal de sortie c, *datemin[c]* soit une borne inférieure de la date d'émission d'un message sur ce canal (le calcul de *datemin[c]* peut faire intervenir une valeur δ associée au comportement du processus simulé - voir Remarque 1 -).
   - La *partie contrôleur*, chargée de ne délivrer les messages à la partie simulation que quand celle-ci à le droit de les consommer (c'est à dire quand un message "plus ancien" ne risque pas d'arriver ensuite).

3. A tout canal entrant c est associée une file de messages : $fdélivrés_c$ où le contrôleur stocke les messages au fur et à mesure qu'ils deviennent consommables. Ces messages sont prélevés par le simulateur. Cette file est illimitée.

C'est l'algorithme du contrôleur qui nous intéresse ici :

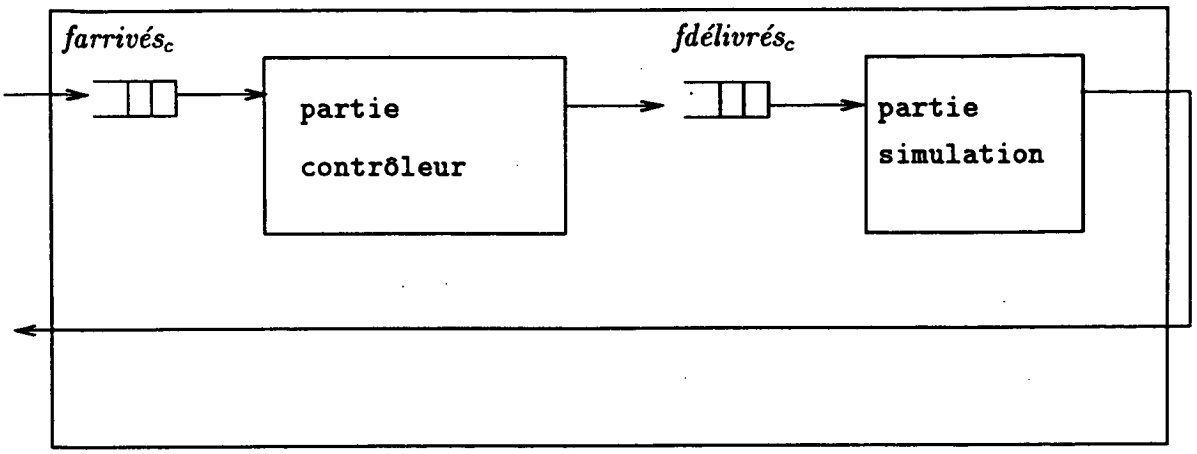

Figure 5 : Décomposition du simulateur d'un processus

Tmin : temps {représentation locale du temps simulé}
estampmin : temps {borne inf des estampilles des messages arrivés}
datemin : [cs]temps {résulat de la procédure *simuler*}

Algorithme du contrôleur :

DEBUT
  REPETER
    simuler(*estampmin*)(*datemin*); {fait progresser la simulation tant que possible}

    POUR TOUT $c \in cs$
    FAIRE
        SI *datemin*[c] > date du dernier message émis sur c
        ALORS émet $(datemin[c], NULL)$ sur c
    FAIT

    POUR TOUT $c \in ce$ FAIRE attendre que c ne soit pas vide FAIT
    $estampmin := \min_{c \in ce}(tempscanal_c)$

    POUR TOUT $c \in ce$
    FAIRE
        POUR TOUT message $(estampmin, m) \in farrivé_c$
        FAIRE
          enlever $(estampmin, m)$ de $farrivé_c$
          SI $m \neq NULL$

ALORS mettre $(estampmin, m)$ dans $f\acute{e}livr\acute{e}_c$ FSI
FAIT
FAIT

FIN
FIN

Remarques :

1. C'est au simulateur de calculer les valeurs δ, réalisant la prévision sur le comportement futur du processus. Si ce calcul peut être simple pour des processus correspondant au modèle simple "file d'attente avec serveur", où δ correspond au temps de service, ce calcul peut être beaucoup plus délicat dans le cas de processus plus généraux. Dans tous les cas c'est le compilateur qui doit produire, au vu du texte source du programme de simulation, le code permettant d'évaluer cette information à l'exécution.

2. On a supposé que la file de réception des messages était infinie, c'est à dire qu'un processus peut toujours émettre un message. Ce n'est évidement pas le cas dans la réalité, mais nous faisons l'hypothèse que ce problème de contrôle de flux dans le réseau est réglé à un autre niveau du système. Dans [14], les auteurs proposent un algorithme qui fait l'hypothèse que les files sont bornées, et intègre donc cet aspect.

3.3.4 Stratégie cohérente a priori avec guérison de l'interblocage

Cette stratégie a été introduite par Chandy et Misra [12]; dans ce cas la simulation consiste à répéter la séquence suivante :

1. Simuler jusqu'à l'interblocage.

2. Détecter l'interblocage.

3. Guérir l'interblocage en relançant l'exécution d'un ou plusieurs processus (remarquons que dans ce cas la guérison ne nécessite pas de tuer un processus comme c'est le cas dans les systèmes d'exploitaton).

Dans la première phase les processus n'émettent que des messages de la simulation, toujours estampillés avec la valeur du temps simulé. Ils font progresser la simulation en itérant sur la consommation du message reçu sur le canal ayant le plus petit temps-canal. Quand le canal ayant le plus petit temps-canal est vide le processus se bloque.

Un processus de contrôle détecte l'interblocage grâce à un algorithme réparti (par exemple celui de Dijkstra et Scholten [17] ou celui de Chandy et Misra [15]).

Quand le processus de contrôle a détecté un interblocage il demande aux autres processus de démarrer un calcul réparti qui permet de déterminer le, ou les processus, pouvant redémarrer sans introduire de violation du principe de causalité.

Il faut noter qu'il existe toujours au moins un processus pouvant reprendre la simulation si on fait l'hypothèse que le modèle, et le programme de simulation correspondant, sont corrects (ie. ne se bloquent pas). En effet, parmi tous les messages en attente sur les canaux entrants de tous les processus, celui qui a la plus petite estampille *Tmin* peut être consommé par son destinataire car plus rien ne peut modifier l'état du système avant la date *Tmin*. L'algorithme de guérison consiste alors simplement à calculer cette valeur *Tmin* ainsi que les destinataires des messages estampillés *Tmin*.

Mais cet algorithme peut être amélioré en remarquant qu'un message estampillé par t' avec $t' < t$ peut être consommé si le graphe des processus est tel que le message estampillé par t ne peut influencer celui estampillé par t'. Ainsi sur l'exemple de la figure 6, où figurent les messages en attente de consommation, le message avec la plus petite estampille est $(5, m_1)$ mais p_3 peut consommer le message $(6, m_3)$ car la prise en compte de ce message ne peut pas avoir de répercussion sur le traitement du message $(5, m_1)$ par p_5. Par contre le message $(8, m_2)$ ne peut pas être traité par p_4, car ce processus peut recevoir de p_3 un message ayant une estampille inférieure. D'une manière générale p_i peut consommer un message (t_i, m) si t_i est inférieur à l'estampille de tout message produit par un processus précedant p_i (p_j précède p_i si il existe un chemin allant de p_j à p_i dans le graphe des processus).

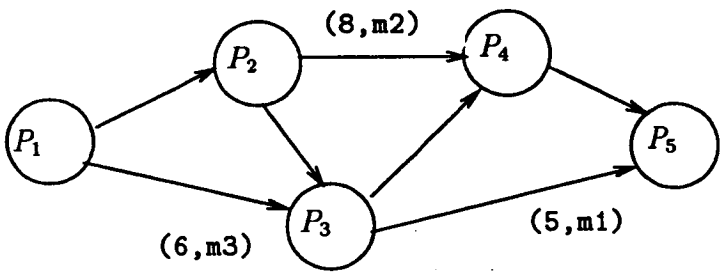

Figure 6 : Réseau de processus interbloqués

Dans [12] et [18] cet algorithme est décrit dans le cas où un processus peut se bloquer en voulant émettre un message (files bornées), nous en donnons ici une description dans le cas simplifié où on considère des files infinies.

A chaque lien (p_i, p_j) on associe les valeurs suivantes :

- T_{ij} = estampille du dernier message envoyé de p_i vers p_j.
- U_{ij} = estampille du prochain message qui sera envoyé de p_i vers p_j, si p_i ne reçoit rien d'autre.
- W_{ij} = borne inférieure de l'estampille des messages qui seront envoyés dans le futur de p_i vers p_j.

P_i connait T_{ij} pour tous ses canaux de sortie, il peut calculer les valeurs de U_{ij} pour tous ses canaux de sortie à partir de son état local actuel. Par contre pour calculer les W_{ij} il doit

coopérer avec d'autres processus. Si D est le diamètre du graphe des processus les W_{ij} peuvent être calculés par p_i comme suit :

ce désigne l'ensemble des canaux d'entrée de p_i
cs désigne l'ensemble des canaux de sortie de p_i

POUR TOUT $j \in cs$
FAIRE
  $W_{ij}^1 := U_{ij}$
  envoyer W_{ij}^1 à p_j
FAIT

POUR TOUT $r \in ce$
FAIRE attendre le message portant W_{ri}^1 FAIT

POUR k DEPUIS 2 JUSQUA D
FAIRE
  $W_{ij}^k := max(T_{ij}, \min(U_{ij}, min_{r \in ce}(W_{ri}^{k-1})))$
  envoyer W_{ij}^k à p_j
  POUR TOUT $r \in ce$
  FAIRE attendre le message portant W_{ri}^k FAIT
FAIT

$W_{ri} = W_{ri}^D$ pour tout $r \in ce$

Après ce calcul p_i peut mettre à jour le temps-canal de chacun de ses canaux d'entrée ($tempscanal_r = W_{ri}$) et reprendre son exécution si il y a un message sur le canal ayant le plus petit temps-canal. L'algorithme ci-dessus est un algorithme distribué à phases. Son but est de calculer explicitement les informations sur le futur proche du système, informations qui étaient implicitement véhiculés par les messages NULL dans l'algorithme précédant.

3.3.5 Une stratégie cohérente a posteriori : le temps virtuel

Cette stratégie ("time warp"), développée par Jefferson [21] est basée sur la notion de temps virtuel. Contrairement aux précédentes elle n'impose aucune contrainte sur l'évolution asynchrone des processus de la simulation : chacun avance à son propre rythme et son horloge locale progresse en conséquence. Lors de l'arrivée d'un message *(t,ms)*, le processus récepteur *pr* fait progresser son horloge jusqu'à *t* et traite le message, ce qui peut provoquer des envois de messages estampillés avec des valeurs supérieures ou égales à *t*.

Un problème surgit lorsque, à la réception du message *(t,ms)*, l'horloge locale de *pr* possède une valeur *t'* supérieure à *t*. Il faut dans ce cas remonter le temps local de la simulation jusqu'à *t*, puis consommer le message *(t,ms)* et recommencer la simulation à partir de *t*. Pour cela le processus *pr* doit "défaire" toutes les actions effectuées entre *t* et *t'*, ce "retour en arrière" peut prendre diverses formes suivant le type d'action en cause :

- Defaire une *action locale* consiste simplement à revenir à un ancien état des variables locales du processus, ancien état que l'on aura donc du sauvegarder.

- Une *émission de message* sera défaite en émettant un "anti-message" vers le même destinataire, avec la même estampille que le message initial. La réception d'un anti-message provoquant chez le recepteur soit l'élimination du message initial s'il n'avait pas encore été consommé, soit un retour en arrière jusqu'à la date correspondant à l'estampille de cet anti-message.

- Les *actions définitives* (ie. sur lesquelles on ne peut pas revenir comme les entrées/sorties), sont différées jusqù à ce que la simulation ait progressé jusqùà un point où on est sûr de ne pas avoir à les défaire.

Cette stratégie nécessite donc de conserver les états successifs des variables locales de chaque processus, la liste de tous les messages émis et reçus. Pour pouvoir libérer de la place en mémoire correspondant à une partie de ces sauvegardes, et pour pouvoir effectuer les E/S on introduit la notion de *temps virtuel global* (GVT). A un instant donné le GVT représente une borne inférieure sur les dates de retour en arrière possibles : toutes les actions effectuées à un temps simulé inférieur à GVT ne seront jamais défaites. Les sauvegardes des états ou messages ayant une date inférieure au GVT peuvent être oubliées, de même les E/S antérieures au GVT peuvent être réalisées réellement. Un algorithme distribué,exécuté à intervalles réguliers, permet de faire progresser ce GVT à partir des horloges locales de chaque processus et des dates d'émission des messages envoyés mais non encore consommés par leur destinataire.

Une amélioration de l'algorithme de retour en arrière à été proposé [22] : lors d'un retour arrière de t à t', au lieu d'envoyer immédiatement un anti-message pour tout les messages émis entre t' et t (soit MES_{anc} cet ensemble), on reprend la simulation à partir de t' en calculant la suite de messages à émettre dans cette nouvelle situation (soit MES_{nouv} cet ensemble). Seuls les messages appartenant à $MES_{anc} - MES_{nouv}$ sont annulés par un anti-message, ceux appartenant à$MES_{nouv} - MES_{anc}$ sont transmis, et les autres sont ignorés (ils ont déja été émis dans la simulation précédente). Un système orienté vers la simulation, basé sur cette stratégie, à été développé sur une machine hypercube [22].

3.4 Liens algorithmiques avec d'autres problèmes

Comme nous l'avons vu les techniques utilisées pour mettre en œuvre un simulateur distribué ont pour but de faire progresser un temps virtuel (le temps de la simulation) de façon cohérente (ie. sans qu'un événement "futur" puisse influencer un événement "passé").

Pour cela la technique "a priori" consiste à ne pas consomer un message m estampillé t sur un canal d'entrée c tant qu'il existe des canaux d'entrée c' dont le temps-canal est inférieur à t. Les messages NULL permettent d'éviter une situation d'interblocage si le canal c' est vide de messages de la simulation. De manière plus générale la solution algorithmique a donc consisté à introduire une *règle* (et à assurer que sa mise en œuvre est correcte des points de vue sureté et vivacité) précisant quand un message reçu peut être délivré (cf. figure 5), le message restant bloqué tant qu'il ne peut pas être délivré.

Le même problème se pose dans la mise en œuvre de la diffusion atomique de messages [9], dans laquelle tous les messages (quels que soient leurs émetteurs) doivent être délivrés à tous les sites dans le même ordre (si l'on considère un contexte où les défaillances de sites sont possibles chaque message doit, de plus, être délivré à tous les sites non défaillants ou à aucun : c'est la propriété d'atomicité).Comme on le voit, lorsqu'un message m est reçu, il ne peut être immédiatement délivré : pour cela il faut être sûr que, relativement à l'ordre total dans lequel les messages sont délivrés, aucun message antérieur à m ne sera reçu (on dit alors que m est *stable*). Il s'agit comme on le voit de la même règle que la précédente. Les solutions proposées s'appuient d'une part sur une technique d'estampillage pour définir l'ordre total (sûreté) [25], et d'autre part sur un protocole qui nécessite deux phases d'échanges de messages pour garantir que les messages seront effectivement délivrés (vivacité) [25],[9],[20],[32].

Un autre problème où l'on a une règle analogue (mais plus simple car la vivacité est garantie a priori), est le contrôle de concurrence fondé sur l'estampillage dans les systèmes transactionnels [11] [5]. Les écritures issues des transactions n'y sont pas réalisées de façon directe : elles sont mémorisées sous la forme de pré-écritures qui seront soient transformées en écritures effectives si la transaction termine, soit éliminées si la transaction est annulée. Les pré-écritures sont estampillées et appliquées à l'objet dans l'ordre des estampilles. Ainsi une pré-écriture estampillée t peut bloquer une écriture estampillée t' si $t < t'$ (sureté). La vivacité est assurée par la progression des transactions elles-même : la pré-écriture estampillée t sera soit annulée soit transformée en écriture (il ne peut pas y avoir d'interblocage).

Remarquons de manière plus générale que le problème du controle de la concurrence a donné lieu, tout comme la simulation répartie, à deux grandes classes de schémas d'exécution : les techniques de controle a priori (utilisant des verrous ou des estampilles) et les techniques de controle a posteriori [24] qui peuvent nécessiter un retour en arrière (ie. de réexécuter certaines transactions).

4 Conclusion

Dans ce rapport on a examiné deux problèmes importants de la simulation distribuée des systèmes à événements discrets : la modélisation de l'application simulée en termes de processus communiquant par messages et la définition de schémas d'exécution répartie. Un simulateur y a été présenté comme un interpreteur de temps virtuel (le temps du programme de simulation). Réaliser un simulateur consiste à définir un schéma d'exécution distribué (distributed run-time) qui assure la progression du temps virtuel (propriété de vivacité de ce schéma d'exécution) sans mettre en défaut les relations de causalité présentes dans l'application simulée (propriété de sureté). Différents types de tels interpréteurs ont été présentés tant dans un cadre synchrone (où la simulation est dirigée par le temps) que dans un cadre asynchrone (où la simulation est dirigée par les événements).

Remarquons que dans le cas où la simulation est dirigée par le temps et où l'interpréteur de temps virtuel est un synchroniseur, la propriété de vivacité est relativement facile à obtenir, alors qu'il n'est pas possible de simuler des réactions de durée nulle dans le temps virtuel (puisque toute interaction consomme une unité de temps). Par contre dans le cas de la simulation dirigée

par les événements il est facile de mettre en œuvre des réactions de durée nulle, mais la difficulté provient de la mise en œuvre de la propriété de vivacité.

Ce dernier type d'interpréteurs distribués de temps virtuel utilise de façon implicite ou explicite une connaissance a priori du futur (look ahead). L'interpreteur utilisant les messages NULL envoie explicitement des messages signifiant une absence d'interaction jusqu'à une certaine date dans le futur. Dans le cas de la technique avec retour arrière (time warp) le retour arrière est une façon implicite d'indiquer une mauvaise progression du temps virtuel et amène donc indirectement une information sur le futur. Cette connaissance sur le futur, nécessaire à la mise en œuvre de la vivacité n'est pas nécessaire lorsque la simulation est dirigée par le temps. Une analyse intéressante de ce phénomène est faite dans [13] en terme de connaissances conditionnelle et inconditionnelle. L'asynchronisme entre les processus qui réalisent la simulation y est d'autant plus grand lorsque les deux types de connaissance sont utilisés.

Signalons enfin que les schémas d'exécution des langages synchrones [10], [26], [7] et [6] peuvent être vus comme des interpréteurs de simulation dans lesquels la progression du temps (réel dans ce cas, et qui est un *signal d'entrée* particulier) n'est pas à construire mais est imposée, et assurée, par l'environnement du programme interprété. Le temps propre à la machine qui supporte le programme synchrone ou la simulation n'a en effet aucune incidence sur le calcul réalisé, d'où les hypothèses de temps de traitement null dans les langages synchrones et de durée de transfert nulle pour les messages dans la simulation dirigée par les événements.

Nous envisageons de poursuivre cette étude de la simulation distribuée selon trois axes de réflexion : premièrement en essayant d'accélérer la stratégie a priori avec prévention de l'interblocage en utilisant d'une manière plus large les échanges d'informations conditionnelles ou inconditionnelles entre processus; deuxièmement en adaptant ces stratégies à l'environnement particulier de la machine parallèle à communication de message disponible à l'Irisa (l'hypercube d'Intel caractérisé d'une part par un nombre de processseurs réduit - 64 dans la version actuelle- par rapport au nombre de processus envisageables dans un modèle à simuler, et d'autre part par un système matériel et logiciel offrant pratiquement un réseau complètement maillé); enfin en essayant de caractériser les types de modèle les plus adaptés à telle ou telle stratégie d'interprétation.

Bibliographie

[1] M. Adam, Ph. Ingels, and M. Raynal. *Algorithmes Distribués synchrones et systèmes répartis asynchrones : concepts, mises en œuvre et expérimentations.* Rapport de Recherche RR-0862, INRIA, Centre IRISA, Rennes, July 1988. 27 p.

[2] M. Adam, Ph. Ingels, and M. Raynal. The meaning of synchronous distributed algorithms run on asynchronous distributed systems. In *The Third International Symposium on Computer and Information Sciences, Izmir*, pages 307–316, November 1988.

[3] B. Awerbuch. Complexity of network synchronization. *Journal of ACM*, 32(4):801–823, October 1985.

[4] R.L. Bagrodia, K. M. Chandy, and J Misra. A message-based approach to discrete event simulation. *IEEE Trans. on Soft. Eng.*, 13(6):654–665, June 1987.

[5] P.A. Bernstein and N. Goodman. Concurrency control in distributed data base system. *Computing Surveys*, 13,2:185–221, June 1981.

[6] G. Berry. The esterel synchronous programming language. *To appear in Sciences of Cmputer Programming*, 1989.

[7] G. Berry and G. Gonthier. Real time programming: special purpose or general purpose languages. In *Proc. IFIP congress*, 1989. invited Talk.

[8] H. Bezivin, J. et Imbert. Une approche pratique à la simulation distribuée. *BIGRE*, (24):17–50, Mai 1981.

[9] K.P. Birman and T.H.A. Joseph. Reliable communication in the presence of failures. *ACM TOCS, vol 5, No 1*, 47–76, February 1987.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre, a declarative language for real-time programming. In *Proceedings ACM Conference on Principles of Programming Languages*, 1987.

[11] S. Ceri and G. Pelagatti. *Distributed Data Bases, Principles and Systems.* Mac Graw Hill, 1984. 393 pages.

[12] K. M. Chandy and J Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. of the ACM*, 24(11), Avril 1981.

[13] K.M. Chandy and J. Misra. *Conditional knowledge as a basis for distributed simulation.* Rapport de Recherche 5251:TR:87, California Institute of Technologie, 1987.

[14] K.M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of ditributed programs. *IEEE Trans. on Soft. Eng., Vol. 5, No 5*, 440–452, September 1979.

[15] K. M. Chandy and J. Misra. Termination detection of diffusing computation in communicating sequential processes. *ACM Toplas*, 4,1:37–43, January 1982.

[16] O .J. Dahl, B. Myhrhaug, and Nygaard. *Simula 67 Common Base Lanugage.* Technical Report, Oslo:Norwegian Computing Centre, 1970.

[17] E. W. D. Dijkstra and C. S. Scholten. Termination detection for diffusing computation. *Information Processing Letters*, 11:217–219, August 1980.

[18] R.M. Fujimoto and D.A. Reed. *Multicomputer networks: message based parallel processing.* The MIT Press, 1987. 380 p.

[19] J.M. Helary and M. Raynal. *Synchronisation et contrôle des programmes et des systèmes répartis.* Eyrolles, 1988. 200 p.

[20] C. Jard and M. Raynal. The rudiments of object distribution in distributed systems. In 2^{th} *Int. Conf. on computers Inf. Sciences, Istambul*, 1987.

[21] D. Jefferson. Virtual time. *ACM Toplas, Vol. 7, No 3*, 404–425, July 1985.

[22] D. Jefferson et al. Distributed simulation and the time warp operating system. In 11^{th} *ACM Symposium on Operating systems principles, Austin, Texas*, pages 77–93, Operating system Review, ACM Press, November 1987.

[23] W.H. Kaubisch, R.H Perrot, and C.A.R. Hoare. Quasiparallel programming. *Software-Practice and Experience, vol. 6, no. 3*, 341–356, July-September 1976.

[24] H.T. Kung and J.T Robinson. On the use of optimistic methods for concurrency control. *ACM TODS*, 6,2:213–226, June 1981.

[25] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications. of the ACM*, 21(7):558–565, July 1978.

[26] P. Le Guernic, A. Benveniste, P Bournai, and T Gauthier. Signal : a data-flow oriented language for signal processing. *IEEE Trans. on ASSP*, 34(2):362–374, 1986.

[27] A.D. Malony, B.D. McCredie, and D.A. Reed. Parallel discrete event simulation using shared memory. *IEEE Trans. on Soft. Eng., Vol. 14, No 4*, 541–553, April 1988.

[28] J. Misra. Distributed discrete-event simulation. *Computing Surveys, Vol. 18, No 1*, 39–65, March 1986.

[29] J.K. Peacock, E.G. Manning, and J.W. Wong. Distributed simulation using a network of processors. *Computers Network*, 3:44–56, 1979.

[30] M. Raynal. *Systèmes répartis et réseaux : concepts, outils et algorithmes.* Eyrolles, Février 1987. (également the MIT Press, 1988).

[31] R. Righter and J.C. Walrand. Distributed simulation of discrete event systems. In *Proc. of the IEEE*, pages 99–113, January 1989.

[32] F.B. Schmuck. The use of efficient broadcast protocols in asynchronous distributed systems. Ph. Thesis, Cornell University, 1988. 124 p.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 470 **PROGRAMMING WITH MALI - THE INTERPRETATION OF PROLOG PROGRAMS**
Louis CHEVALLIER, Serge LE HUITOUZE, Olivier RIDOUX
16 Pages, Mai 1989.

PI 471 **VERS UNE PROBLEMATIQUE DE L'ALGORITHMIQUE REPARTIE**
JeanMichel HELARY, Michel RAYNAL
12 Pages, Mai 1989.

PI 472 **DERIVATION SYSTEMATIQUE D'UN ALGORITHME DE SEGMENTATION D'IMAGES - UN EXEMPLE D'APPLICATION DU FORMALISME GAMMA**
Christian CREVEUIL, Gersan MOGUEROU
46 Pages, Mai 1989.

PI 473 **MICROCODE OPTIMIZATION FOR THE PCS PROCESSOR**
François BODIN, François CHAROT, Charles WAGNER
26 Pages, Mai 1989.

PI 474 **ALGEBRAICALLY CLOSED THEORIES**
Eric BADOUEL
22 Pages, Mai 1989.

PI 475 **QUELQUES OUTILS GRAPHIQUES POUR LA MODELISATION DU CONTROLE D'EXECUTION EN ROBOTIQUE DE COOPERATION**
Jean-Christophe PAOLETTI, Lionel MARCE
52 Pages, Juin 1989.

PI 476 **SIMULATION REPARTIE DE SYSTEMES A EVENEMENTS DISCRETS: PARTIE 1 : MODELISATION ET SCHEMAS D'EXECUTION**
Philippe INGELS, Michel RAYNAL
26 Pages, Juin 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique